Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 465–474

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

ParaView + Alya + D8tree: Integrating High Performance Computing and High Performance Data Analytics

Antoni Artigues[1], Cesare Cugnasco[12], Yolanda Becerra[12], Fernando Cucchietti[1], Guillaume Houzeaux[1], Mariano Vazquez[13], Jordi Torres[12], Eduard Ayguadé[12], and Jesus Labarta[12]

[1] Barcelona Supercomputing Center {cesare.cugnasco,yolanda.becerra,fernando.cucchietti, guillaume.houzeaux,mariano.vazquez,jordi.torres, eduard.ayguade,jesus.labarta}@bsc.es,antoni.artigues.feina@gmail.com
[2] Universitat Politècnica de Catalunya - Department of Computer Architecture - Spain
[3] IIIA CSIC, Bellaterra - Spain

Abstract

Large scale time-dependent particle simulations can generate massive amounts of data, making it so that storing the results is often the slowest phase and the primary time bottleneck of the simulation. Furthermore, analysing this amount of data with traditional tools has become increasingly challenging, and it is often virtually impossible to have a visual representation of the full set.

We propose a novel architecture that integrates an HPC-based multi-physics simulation code, a NoSQL database, and a data analysis and visualisation application. The goals are two: On the one hand, we aim to speed up the simulations taking advantage of the scalability of key-value data stores, while at the same time enabling real-time approximated data visualisation and interactive exploration. On the other hand, we want to make it efficient to explore and analyse the large data base of results produced. Therefore, this work represents a clear example of integrating High Performance Computing with High Performance Data Analytics. Our prototype proves the validity of our approach and shows great performance improvements. Indeed, we reduced by 67.5% the time to store the simulation while we made real-time queries run 52 times faster than alternative solutions.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: HPC, HPDA, Key-Value data stores

1 Introduction

High performance simulations can run on thousands of computers for several hours and generate massive quantities of data, such as the position and properties of particles at each time step. Large scale simulations track hundreds of millions of particles, and the size of the output files containing all this information can easily be in the order of Terabytes.

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.170

Traditionally, simulation results are stored in one or more files in formats such as CSV, HDF5 or netCDF. Storing the data can be a slow process, in some cases becoming a major performance bottleneck. Moreover, in a Supercomputing environment applications do not usually access the locally attached disks, instead they use instead a distributed file system such as IBM GPFS [2]. However, as described in Section 2, writing parallelly into a single file can be challenging and often requires additional collective synchronisation and communication between processes, which can limit performance. We find this approach suboptimal for three main reasons: 1. It does not exploit data locality, thus requiring data to be transmitted over the network many times, 2. it requires worker synchronisation to optimise the I/O, 3. it does not allow to analyse the results while the simulation is still running.

In this paper, we investigate an alternative approach that employs a distributed key-value database using the local disks. Thus, not only we reduce network usage, but we can index the simulation in real time and allow interactive exploration either at run time or off-line. To meet these requirements, we adopted and extended a distributed NoSQL database, which ensures consistency, availability, and high concurrency while maintaining the data set indexed.

As we approach the exaflop scale and simulation data grows in size, the future of scientific visualization hinges not only on more powerful hardware, but also on efficient algorithms that can reduce the precision of visualisation while maintaining the statistical proprieties of the data. To address this issue we implemented Qbeast, a distributed peer-to-peer system that extends the D8tree [8]– a multidimensional index that, on the one hand, allows efficient sampling, and on the other hand, maintains indexes in real-time under heavy write loads. Our aim is to give the user the possibility of choosing the right trade-off arbitrarily between level of precision and response time. Our system allows researchers to visualize simulations in real time and after the simulation has finished, it allows visualizations of complex queries as well. For example, what is the path followed by a particular particle, how do particles mix in a region, where do particles in this region come from, how many particles go across a given section, and all options that can be programmed into a DB query.

The paper is organized as follows: Section 2 describes related work, while Section 3 includes background information. Section 4 discusses the implementation choices taken for our real-time multidimensional index, while Section 5 describes the overall prototype design. Section 6 reports all the experiments we carried out on our architecture and in Section 7 we draw our conclusions and discuss future lines of research.

2 Related work

Nowadays, high performance computer simulations run on thousands of parallel cores and generate outputs of the order of TeraBytes. How to parallelly store, organise, and analyse these results has become a key factor for both performance and usability. Traditionally, the output with the results is stored in files. There are plenty of different formats, with some of the most popular being HDF5[10] and netCDF[16]. Although their data models differ, since version 4, netCDF uses HDF5 as the backend, and thus they have similar performance characteristics. The problem with file storage is how to achieve high performance while thousands of concurrent processes are accessing in parallel. A straightforward approach is to generate an output file for each process, but this then requires having thousands of files which are complex to manage and analyse. Alternately, HDF5 allows creating data set partitions – hyperslabs – that can be written and read independently by different processes. HDF5 offers two MPI-based accessing modes: independent and collective. In the first case, all processes can access the file without any synchronisation, but this can cause several random I/O calls and thus penalises performance.

Collective access reduces the I/O by assigning a subset of MPI tasks to act as "aggregators" so that they can gather smaller and independent requests in larger and contiguous I/O accesses.

Whether to use an approach or another is strongly influenced by the data layout and the distributed filesystem[14][17] used. For example, if each process writes into a hyperslab that maps to adjacent disk partitions, there is no need for collective I/O. However, this is not the case for particle simulations. The typical dimensions of particle simulations are time step, position, and particle identifier. However, each process simulates all the particles that fall into its sub-domain at a given time step. Therefore, each process writes into randomly distributed positions. Indeed, libraries specifically tailored for particles simulations, such as H5hut[13], heavily employ collective I/O.

An alternative approach is the one that Alya implements when writing Computational Fluid Dynamic in HDF5[9]: each process writes into a different dataset inside the same file. In this manner we can achieve good I/0 performance, but the parallelization of the simulation fully dictates the file layout.

In a previous contribution[7], we proposed the integration of ParaView with a map-reduce Hadoop environment. ParaView is one of the most common applications used for analysing the results of a simulation. Once the amount of data exceeds a single computer resource, the only way to visualise a simulation is to use Paraview's distributed render which can use hundreds of CPUs and/or GPUs. As a result, visualising a large simulation might be inconvenient in terms of complexity, time, and cost. However, in large scale simulations it is seldom necessary to visualise all the data at once, let alone for interactive analyses or particular queries. Because of this, we proposed to reduce the amount of data sent to the user by using a map-reduce approach to filter the results and perform *ad-hoc* analysis. For instance, selecting only the trajectory of a single particle, or finding the origin of particles that got stuck in a given area, or computing aggregations such as calculating the percentage of different elements. This approach proved to be valid, but still has some drawbacks: 1. It was done in a post-processing step, so we had to wait for the end of the simulation to upload the results to HDFS, 2. as the data is not indexed, each query requires processing the whole simulation in memory.

An interesting alternative to the approach of storing a smaller resolution version of the data, tailored for visualization, is Paraview Cinema [6, 5]. The philosophy behind Cinema is that beyond certain scales, it is more efficient to store images with all possible combinations for a visualization (distance, angle, variable, etc.) than the actual data needed to produce those images. However, if you need to recover the full data set you need to re-run the whole simulation which could arguably be cheaper, although not necessarily more convenient. With our approach instead, we allow powerful analytic queries and filters that can go beyond what is possible with an only-visualization approach.

3 Background

ParaView[3] is an open-source data analysis and visualisation application. It allows exploring data either via an interactive 3D interface or with a batch approach that employs distributed processing to achieve high-resolution. The ParaView design allows adding plug-ins to implement new filters. Indeed, we created a plug-in to integrate our new functionalities into the ParaView data analysis pipeline.

Alya is the BSC in-house HPC-based multi-physics simulation code [18], designed to simulate highly complex problems and efficiently run on high-end supercomputers. Alya targets engineering problems and thus it can be applied in several domains: the aerospace and automotive industry, biomechanics and biomedical applications, environmental research, and oil, gas

and wind energy fields.

The physical problem that motivated our development is the simulation of air flows through the respiratory system. This biomedical problem ran with Alya [18] in the Julich supercomputer. Of particular medical interest is the simulation of Lagrangian particles transported by fluids, that can be either medicine or pollutants in the air. Powerful tools are needed to analyse the results with the aim of improving medicine diffusers and inhalers. The initial set up we worked with consisted of Alya writing each particle properties in a persistent system: Time step, position, velocity, type, family, etc. A typical simulation generated 300 Gb of plain text (CSV) data, but this is an artificial limit due to practical considerations. We used two physical modules of Alya to carry out this work. The incompressible Navier-Stokes equations are solved in a sniff condition, using a mesh of 350M elements, while Lagrangian particles are transported by fluids, by means of drag law and Brownian motion. All the details concerning the algorithms used in this work can be found in Refs [12, 11, 18].

D8tree [8] is a novel index algorithm designed to support both analytical and data-thinning queries on multidimensional data while aiming to lower latency and achieving linear scalability. The main idea of the D8tree is to merge multidimensional indexing, data sampling and denormalization techniques to achieve high scalability, availability, and performance. The original version has a batch design as it aims to build indexes of completed simulations. As we aim to analyse the results of a simulation while it is running, we implemented a new low-latency system able to ingest and index the mole of data as fast as it is generated.

4 Real-time D8-tree index creation

We previously presented the D8-tree for static datasets[8]: it used a batch processing approach to build the index, which is efficient but it requires the simulation to have completed. To analyse a simulation at run time, we have to build the index at the same speed the data is generated. The D8-tree structure contains copies of the same item at multiple levels of the tree. For each level k, the whole domain is decomposed in 8^k partitions that we call cubes. Each cube has a maximum capacity of C. When a cube reaches C, it sorts its content by a random priority and then it discards the items with lower values. Therefore, a query that starts with the root leaf reads a uniform random sample of data. While descending the tree it can increase the number of samples. Maintaining updated such a structure on disk requires sorting all the elements in each cube, which has a prohibitive performance cost under a continuous stream of insertions. Given an index of K levels, for each insertion, we should access K different cubes, and modify each cube index by adding the new element and possibly removing the overflowing ones.

Such a mutable structure has major drawbacks: it causes high cache pollution and it requires costly synchronisation in multicore architectures. Furthermore, it performs badly with both rotational disks and flash drives. In rotation HDD, the time for moving the magnetic heads to the desired disk partition –known as seek time– is a significant part of the overall time, and thus it penalises accessing to non-consecutive disk sectors. At the same time, as several articles pointed out[4][15], the SSD devices degrade

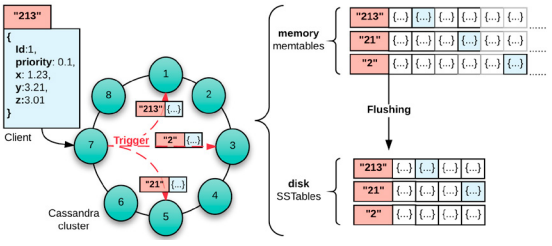


Figure 1: Dynamic D8-tree indexing

under a heavy load of random writes.

LSM-tree indexes, such as the one used in Cassandra, have been pointed out as an alternative to the widely adopted B-tree as they do not suffer of write amplification[19]. LSM-tree designs exploit the hierarchy between the volatile and secondary memory, using the first to optimise accesses to the second. Elements are kept in memory until a threshold is reached and then the items are stored ("flushed") into a disk file in a single sequential write. A background process, called *compactions*, aggregates single files in larger ones.

For all the cited reasons, we decided to follow the LSM-tree approach to create our dynamic D8tree. We modified the database Apache Cassandra to implement our index. We made two changes: we added a trigger and we altered the logic of the flushing algorithm. Figure 1 shows the overall architecture. On the right, we can see that the client sends the request to the designed node where the trigger replicates the query to the higher cubes. Cassandra stores each insertion into an in-memory structure called Memtable keeping the records ordered by their priority. When a Memtable reaches its threshold size, we modify the *flushing* algorithm to write only the first elements to disk. In particular, Figure 1 shows how the particle we inserted went in different positions of the three cubes so that in the higher one "2", it is discarded when flushing the content into an SSTable on disk. Meanwhile, the compaction process takes care to unite the small SSTables into larger ones, thus reducing the index size.

5 Architecture

This section provides a brief summary of how we integrated Alya with Qbeast and it describes how we connected Qbeast with ParaView, allowing to query the system in real-time.

The original Alya architecture is shown in Figure 2a: a single master node manages the simulation output. The Alya master collects all particles from the workers at each time step, and then it appends the new results into a CSV file which is stored on a distributed file system. While the CSV format has inarguably advantages given its simplicity and human-readability, sending all results to one single worker is a major performance bottleneck. We tested two different integration prototypes: master-slave (Figure 2b) and peer-to-peer (Figure 2c).

Writing a high-performance communication protocol can be challenging, thus we decided to connect Alya to Qbeast using the official Cassandra driver. We created the *Alya-Qbeast connector* as a C++ library wrapper for Alya, which is written in Fortran 90. The connector uses the C++ Datastax [1] driver to connect through an asynchronous protocol to Cassandra. The driver manages failover and reconnection in case of messages lost or node crashes, and it takes care of delivering the request to the correct node. Cassandra takes care of persisting, indexing, and efficiently writing to disk all the parallel requests sent by Alya. We used the same Alya-Qbeast connector for both implementations: in the first version, only the Alya master invokes the connector while in the second one each slave uses the connector independently.

As Cassandra takes care of reordering the results, we can avoid the gathering operation in the Alya Master. Figure 2c(c) shows the peer-to-peer architecture. In the master-slave design the master dictates the output time. However, in the peer-to-peer version, the slowest node limits the total output time. Each Alya worker simulates a sub-domain of the whole simulation volume, and therefore it is common that in some phases of the simulation there are workers with a higher number of particles, which consequently need more time to complete the simulation step. In the experiments section, we present a detailed comparison between these two Connector implementations.

The C++ Cassandra driver uses an asynchronous protocol thus allowing to efficiently send several requests to Cassandra in parallel. However, as issuing too many requests leads to system

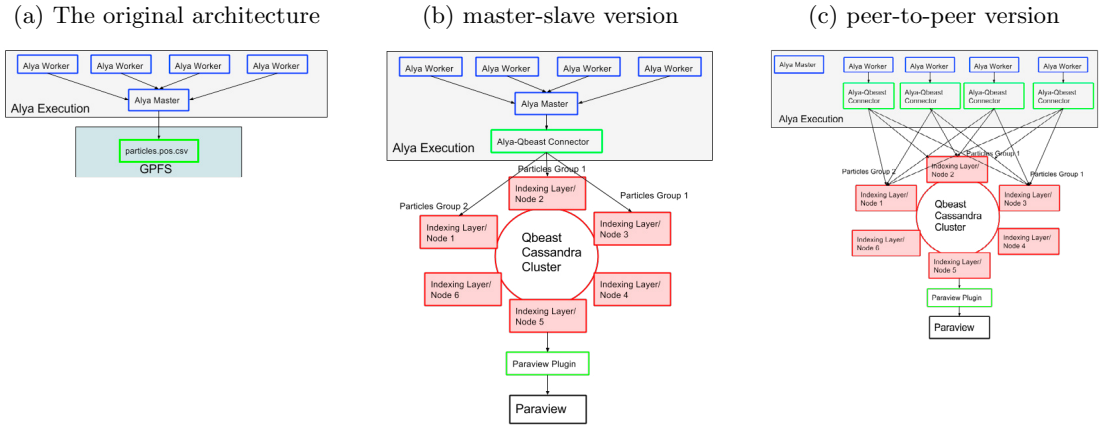


Figure 2: The three different architectures we tried in this article: (a) The original set up where an Alya master node receives and writes all the information, (b) The Alya master node is connected to QBeast nodes, and (c) all Alya workers push information to QBeast nodes independently.

instability and performance degradation we added the parameter **parallelism-level** to limit the number of operations that are in execution at the same time. Another option is to reduce the number of queries by grouping them in larger ones, called batch statements. The trade-off is between network bandwidth and latency. As this aspect influences performance, we defined the parameter **batch-size** that indicates how many requests group into a batch statement.

The overall system is composed by: 1. ParaView 2. the ParaView-Qbeast plug-in 3. the Qbeast Query Engine: a distributed middle layer that queries the D8-Tree index exploiting the Cassandra data locality. 4. the Cassandra nodes: store and serve requests on the D8-tree index.

Usually, ParaView runs on personal computers and renders the data locally, so that if too many particles are loaded together the system collapses. While it is possible to distribute the rendering on multiple machines, it is costly both in terms of time and resources. However, in many cases a random sample of the results is enough for an interactive analysis of the simulation: we allow the user to choose the right trade-off between the level of detail and system responsiveness. By tuning the *precision* and *max-results* parameters, the user defines the percentage of data to visualize or sets an upper bound to the number of elements to fetch. Thanks to the D8-tree [8], this can be efficiently implemented with Cassandra.

The process of sampling a dataset, also called data-thinning, would normally require the following steps: 1. Read the whole data set 2. Randomly discard in memory 99 elements every 100 3. return the 10000 filtered element. In such a way, reading the whole dataset or just a sample requires almost the same time.

In contrast, the D8-Tree has a response time proportional to the number of elements returned: getting one random part out of ten requires one tenth of the time to read the whole data. The D8-tree uses a random priority to organize the data into cubes, so that in the higher cube we find the elements with higher priority. Therefore, if we want to get the k% of a dataset we just have to: 1. read the root cube 2. select all elements with $priority \leq k$ 3. if point 2 returns new items, we descend into the 8 cube children and for each of them we go back to point 2. 4. if all queries have completed, we return the results to the client.

In Figure 3 we show some screen-shots taken during the simulation of particles flowing

through a human nose. With this visualizations we were able to obtain insights about how the simulation was proceeding before it completed.

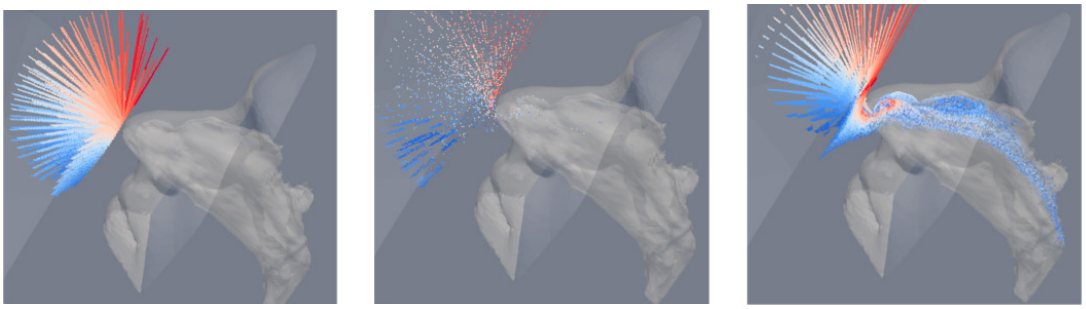


Figure 3: Screen shots of real-time visualization of particles flowing into the respiratory system in a rapid air intake simulation.

6 Experiments

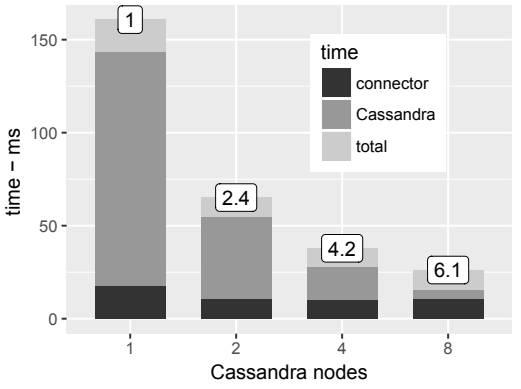
In this section we present the tests we carried out on our Alya-Qbeast integration. We analyse its performance compared with the original Alya implementation, and we discuss the parameters that mostly influence performance. All tests simulated the same particle respiratory system and ran on the BSC-CNS Marenostrum 3 Supercomputer. Each node in Marenostrum 3 is equipped with two Intel SandyBridge-EP E5-2670 20M 8-core for a total of sixteen cores at 2.6 GHz processor base frequency and 32 Gb of DDR3-1600 DIMMS ram. Each node uses the IBM GPFS file system [2] running on the Infiniband FDR10. Both Alya and Cassandra started at the same time in the supercomputer: Cassandra employs an internal BSC's library module, that allows creating a Cassandra cluster using a queue job system.

Horizontal system scalability The first experiment tests the scalability of Cassandra: we increased the number of Cassandra nodes to measure the relative performance improvement. We simulated 6.75 million particles moving during 10 time steps using 256 Alya MPI workers. We disabled the Qbeast indexing, and we used the peer-to-peer version.

For each Cassandra cluster configuration, we had to properly configure the Connector. Indeed, to fully take advantage of the additional nodes, we need to increase the parallel insertions. For example, with 4 Cassandra nodes, the best performance is with 200 concurrent requests.

Figure 4a shows that from 1 to 4 nodes, Alya scaled perfectly, in fact the response time halved when the Cassandra nodes doubled. However, with 8 Cassandra nodes, the relative speed-up resulted in only 6 times, instead of the ideal 8. To understand what limits performance, we broke down the output time into Connector and Cassandra. The Connector only includes the time to transform a particle from the Alya format into Cassandra one. The latter considers the database processing time. Figure 4a shows the different time components of an insertion and how they change when increasing the number of Cassandra nodes. We can see the time taken by Cassandra decreases while the time required by the Connector remains stable: with 8 nodes the first reduces of 9.2 while the second improves of only 1.6. With more than 4 nodes the Connector is the major performance bottleneck and it does not allow the system to scale. Future work will improve the connector implementation and help reduce this fixed performance cost.

(a) Cassandra scalability details



(b) GPFS VS Cassandra output time

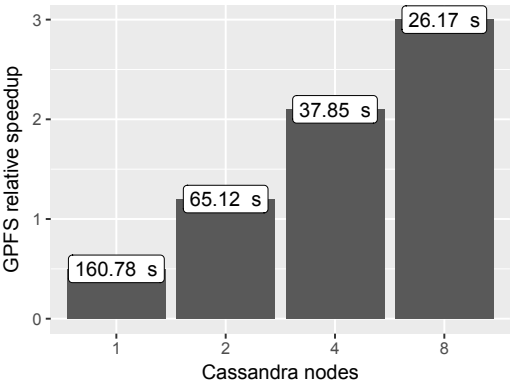


Figure 4: System scalability

GPFS versus Alya-Qbeast This test compares the performance of the original Alya with the peer-to-peer Alya-Qbeast version. The original Alya design has a master process that collects the results from all workers, and then it writes the results into a file on GPFS.

The test simulated 6.75 million particles, 256 Alya MPI, 10 time steps. The original Alya took 80.6 seconds to complete the output step of one iteration. By enabling the Connector and using only one Cassandra node, the output step took 160.78 seconds, while increasing the number of Cassandra nodes reduced the response time to 65 seconds with 2 nodes and to just 26 seconds with 8 nodes. In Fig. 4b we show how Alya runs 3 times faster thanks to Cassandra's linear scalability.

Alya master-slave versus peer-to-peer connector This experiment compares the master-slave and the peer-to-peer Alya connector versions: both write data to Cassandra. The aim is to understand how much the original Alya architecture is penalised by the master-slave approach. For this experiment we considered about 300 thousand particles, simulating their flow during 10 time steps. Alya used 64 MPI workers and insertions were handled by a single Cassandra node with the Qbeast trigger enabled. The D8tree maximum depth was 5.

To fairly compare the two implementations, we had to find the optimal Connector configuration for the parameters *batch-size*(optimal value 5) and *parallelism-level*(optimal value 10). With the optimal settings, the Alya-Qbeast peer-to-peer version doubles the performance of the master-slave version. Even though we did not perform extensive tests of all possible settings, we had the possibility to estimate that a master-slave approach requires about 84% more than the peer-to-peer one.

Dynamic indexing overhead In order to index simulation data in real time, the Qbeast trigger needs to duplicate each insertion multiple times so that it propagates up to the higher tree levels. To measure its overhead, we ran a new test with the same condition of the previous scalability test but enabling the indexing. With the D8tree maximum depth set to 5, the execution took 105.7 seconds. Compared to 37.85 seconds of the non-indexed version, it proved to be 1.8 times slower. As expected, indexing on real-time added an overhead: as maximum depth equalled 5, Qbeast replicated each insertion 5 times. However, we experienced a smaller performance detriment, as long as the Qbeast periodically filters in memory the inserts that would not fit into the index, thus dramatically reducing the amount of data written to disk and decreasing the algorithm overhead from 5 to 1.8.

Query performance experiment This last test compares the query capability when using the D8-tree or when storing data into CSV files. For our experiment, we chose an important

query in our visualisation system: A small sampling over a large simulation area. We decided to perform a query that returns 1.5% of the particle present in the whole space domain, over a dataset of about ten million particles. The simulation considered 54000 particles during 200 time steps. We used one Cassandra node and 12 Alya nodes connected through MPI. We ran the particle simulation with the trigger enabled to generate the D8-tree index, and we used our ParaView Plug-in to query and visualise the random sample of particles. We obtained the response in 4.19 seconds.

ParaView does not allow this kind of query so we used Apache Spark to read and filter on memory the results from the Cassandra database to compare. Apache Spark required about 210 seconds to load and filter the sample in memory, while with our system we were able to retrieve and visualise the results in only 4.19 seconds. This massive speed-up –52 times faster– enables the user to analyse interactively a simulation.

7 Conclusion and future work

In this paper, we presented an innovative prototype architecture that integrates Alya, an HPC physics simulation software, Qbeast, a distributed system for real-time multidimensional indexing and data-thinning, and ParaView, an analysis and visualisation application.

The aim of our work is to improve the storage of simulations, boosting performance and analytical capabilities. Our tests demonstrated that key-value databases are a viable alternative to plain file storage for simulation persistence, as they allowed to improve the write performance (in one case by up to 65.7%) by simply adding more database nodes. Also, we showed that it is possible to maintain at real-time an index over the simulation results so that it is feasible to visualize the early results of a running simulation. Indeed, while in our prototype the indexing slows down the execution by 31%, it enables extremely fast arbitrary-approximated query on the simulation. Compared with Apache Spark, we achieved a 52 factor speedup.

As the system is an early prototype, there are several possible areas of improvement, such as co-allocating Alya and Qbeast and reducing the overhead of indexing and data transmission.

As a last conclusion and pointing towards the future, the following remarks are worthwhile: (**a**) This paper presents a useful application where High Performance Computing is integrated with High Performance Data Analytics in such a way that it would apply to any kind of complex simulations (and not just particles, or the respiratory system). (**b**) By design, the proposed software architecture allows the use of *ad-hoc* hardware architecture: an HPC cluster where large-scale simulations are run, an HPDA cluster where a distributed data base can be deployed, fed, and analysed, and a client to query the data base. (**c**) This strategy can be easily extended to analyse not just one simulation but a large set of simulations, with transversal queries. For instance, the database is fed by a set of simulations of respiratory systems of a large number or patients under a large number of conditions. Off-line, the user can ask questions like "How are these 3 different designs of inhalers working for girls between 12 and 14 years old, from an average Japanese population, suffering from asthma of these 4 different kinds?". (**d**) An additional layer of machine learning can be easily integrated in the software infrastructure.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 720270 (HBP SGA1). It is also partially supported by the grant SEV-2011-00067 of Severo Ochoa Program, the TIN2015-65316-P project, with

funding from the Spanish Ministry of Economy and Competitivity, the European Union FEDER funds, and the SGR 2014-SGR-1051.

References

[1] C/C++ Driver for Apache Cassandra . `https://goo.gl/Q5pq5sr`.

[2] Ibm general parallel file system. `https://goo.gl/yWmVXw`.

[3] Paraview software. `http://www.paraview.org`.

[4] Agrawal, Prabhakaran, Wobber, Davis, Manasse, and Panigrahy. Design Tradeoffs for SSD Performance.

[5] James Ahrens, S Jourdain, P OLeary, J Patchett, David H Rogers, and M Petersen. In situ mpas-ocean image-based visualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*, 2014.

[6] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H Rogers, and Mark Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434. IEEE Press, 2014.

[7] Antoni Artigues, Fernando Cucchietti, Carlos Tripiana Montes, David Vicente, Hadrien Calmet, Guillermo Marín, and Mariano Vazquez. Scientific Big Data Visualization: a Coupled Tools Approach. 2014.

[8] Cesare Cugnasco, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. D8-tree: A De-normalized Approach for Multidimensional Data Analysis on Key-value Databases. 2016.

[9] Raúl de la Cruz, Hadrien Calmet, and Guillaume Houzeaux. Implementing a XDMF / HDF5 Parallel File System in Alya. pages 1–8.

[10] Folk, Heber, and Koziol. An overview of the HDF5 technology suite and its applications.

[11] Guillamume Houzeaux, Raúl de la Cruz, Herbert Owen, and Mariano Vázquez. Parallel uniform mesh multiplication applied to a navier-stokes solver. *Computers & Fluids*, 80:142–151, 7.

[12] Guillaume Houzeaux, Michel Aubry, and Mariano Vázquez. Extension of fractional step techniques for incompressible flows: The preconditioned orthomin(1) for the pressure schur complement.

[13] Howison, Adelmann, Bethel, Gsell, Oswald, and Benedikt. H5hut : A High-Performance I / O Library for Particle-based Simulations. 2010.

[14] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, and John Shalf. Tuning HDF5 for Lustre file systems. *IASDS '10 Proceedings of the Workshop on Interfaces and Abstractions for Scientific Data Storage*, 5, 2012.

[15] Huu and Haas. The Fundamental Limit of Flash Random Write Performance.

[16] Li, Liao, Alok, Ross, Thakur, Gropp, Latham, Siegel, Gallagher, and Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. *Supercomputing*.

[17] Prost, Treumann, Hedges, Jia, and Koniges. MPI-IO / GPFS , an Optimized Implementation of MPI-IO on top of GPFS. (November 2001):0–14.

[18] Mariano Vazquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmín Aguado-Sierra, Ruth Arís, Daniel Mira, Hadrien Calmet, Fernando Cucchietti, Herbert Owen, Ahmed Taha, and José M. Cela. Alya: Towards exascale for engineering simulation codes.

[19] Wang, Sun, Jiang, Ouyang, Lin, Zhang, Cong, and Jason. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD.